Querying Multiple Tables via Joins

Normalization and Basic Database Design:

In the majority of the examples throughout these lessons, we have queried against views instead of database tables. Views are specifically designed to simplify the code required to get commonly requested information – like consolidated information about employees or customers. Tables in most transactional databases are not designed with this consolidation in mind, rather they are designed with a concept of normalization in mind. This normalization, which we will discuss momentarily, reduces the amount of redundancy and repetition in the data stored within a table. These transactional databases are typically designed to allow for improved write operations: inserting and modifying data. We are interested in read operations – returning data – when we write and execute our SELECT statements.

Systems designed for transactions typically require a SQL user to query against multiple tables at once to return the information they are looking for. Up to this point, we have only focused on how to return data from a single table or view. Using a concept called joins, we will be able to return columns from multiple tables in the same SQL SELECT statement.

If you remember back to the first video lesson, we discussed a few key database concepts. There are two in particular that will be necessary to understand before talking about joins: primary keys and foreign keys. Before going any further, re-read the definitions of each.

In the introduction to this section, a concept called **normalization** was mentioned. Microsoft's Developer Network (MSDN) defines normalization as "the process of refining tables, keys, columns, and relationships to create an efficient database". Let's take a look at an example which will best explain this concept.

Assume we have a table, named Book, that has information about books. Each row contains information about the authors, publishers and information about the book itself. Below might be an example of what this table looks like:

Title	ISBN	Publish Date	Publisher	Author 1	Author 2	Author 3
Book 1	123456789	June 1, 2008	Publisher 1	Author A	Author B	
Book 2	124356798	September 12, 2009	Publisher 2	Author B	Author C	
Book 3	873781927	April 6, 2011	Publisher 3	Author A	Author C	Author D
Book 4	826389163	January 4, 2010	Publisher 1	Author B	Author D	Author E

Book 5	129471352	December 12, 2013	Publisher 2	Author D		

There are a few important details to notice in this table. First, the same publishers and authors are repeated multiple times throughout this table. Most books have multiple authors and certainly several authors write multiple books. There exists significant repetition in each row of this table. This table is known as a denormalized table – that is, repetition and redundancy exist.

If we wanted to normalize this table, we would look at the key areas of redundancy. First, a publisher is the publisher for multiple books. However, there is no more than one publisher associated with each book. The first step in normalizing this table would then be to set up a publisher table. The table would look like:

PublisherID	Publisher Name
1	Publisher 1
2	Publisher 2
3	Publisher 3

The first thing you will most likely notice is the PublisherID column. In a normalized database (most databases in fact), a key or ID column will be created. This key is usually an integer and, in this example, the PublisherID represents the primary key of the Publisher table. For many reasons, including query performance and uniqueness identification, ID columns are almost a near necessity.

Getting back to our normalization example, since we have created a Publisher table with a key assigned to each publisher, let's go ahead and replaced "Publisher 1" with the PublisherID of 1, "Publisher 2" with the PublisherID of 2, and "Publisher 3" with the PublisherID of 3. So, our main table, Book, now looks like:

Title	**ISBN**	**Publish Date**	**Publisher**	**Author 1**	**Author 2**	**Author 3**
Book 1	123456789	June 1, 2008	1	Author A	Author B	
Book 2	124356798	September 12, 2009	2	Author B	Author C	
Book 3	873781927	April 6, 2011	3	Author A	Author C	Author D
Book 4	826389163	January 4, 2010	1	Author B	Author D	Author E
Book 5	129471352	December 12, 2013	2	Author D		

The value in the publisher column represents a row in the Publisher table. This Publisher column in the main book table can now be thought of a foreign key. That is, the Publisher column in the Book table has a foreign key relationship with the PublisherID column of the Publisher table.

Moving on, let's handle the redundancy relating to the authors. The relationship between authors and books is what is known as a "many-to-many" relationship. This is because an author can be related to more than one book, while a book can be related to more than one author. The relationship between publishers and books, on the other hand, was known as a "one-to-many" relationship. That is because one publisher can be related with more than one book, but a book can only be related to a single publisher.

To handle the "many-to-many" relationship that we have between authors and books, we start just like we did with the publishers. We will create an Author table that contains two columns: AuthorID and Author Name:

AuthorID	**Author Name**
1	Author A
2	Author B
3	Author C
4	Author D
5	Author E

We will replace each author with the associated AuthorID from the Author table. We will also create a BookID column for our Book table to follow proper form. The Book table now looks like:

BookID	**Title**	**ISBN**	**Publish Date**	**Publisher**	**Author 1**	**Author 2**	**Author 3**
1	Book 1	123456789	June 1, 2008	1	1	2	
2	Book 2	124356798	September 12, 2009	2	2	3	
3	Book 3	873781927	April 6, 2011	3	1	3	4
4	Book 4	826389163	January 4, 2010	1	2	4	5
5	Book 5	129471352	December 12, 2013	2	4		

This still isn't quite a completely normalized table even though we now have a foreign key relationship on Author 1, Author 2, and Author 3 to the AuthorID column of the Author table. For "many-to-many" tables, a bridge table is often need to bridge the gap between the two previous tables. We are going to create this bridge table by having a table that contains two columns: BookID and AuthorID. Our goal is to have the complete list of combinations of BookID and AuthorID that we see in our Book table represented in this new bridge table. This new bridge table, which we will call BookAuthor, will look like:

BookID	**AuthorID**
1	1
1	2
2	2
2	3
3	1
3	3
3	4
4	2
4	4
4	5
5	4

The primary key of BookAuthor, the column or columns that define uniqueness for the table, will be the combination of BookID and AuthorID. Notice how each combination is distinct; this details is important and will become more apparent when we start practicing joins with SQL statements.

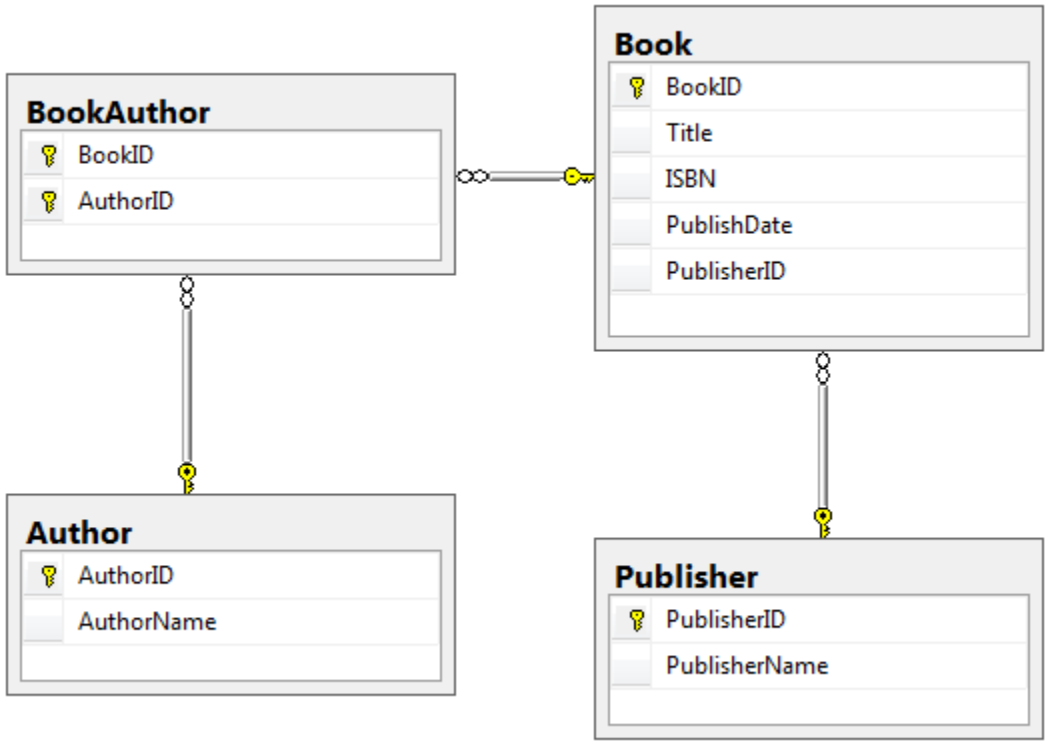
Since we have a bridge between the book and author(s) established, our Book table can be trimmed down to:

BookID	**Title**	**ISBN**	**Publish Date**	**Publisher**
1	Book 1	123456789	June 1, 2008	1
2	Book 2	124356798	September 12, 2009	2
3	Book 3	873781927	April 6, 2011	3
4	Book 4	826389163	January 4, 2010	1

5	Book 5	129471352	December 12, 2013	2

Why can this be done? All of the basic author details are stored in the Author table. All of the information connecting each author to each book is stored in the BookAuthor bridge table. We have eliminated the necessary redundancy in storing multiple columns of author details in the Book table.

We now have a completely normalized database of tables. A database diagram is an effective way to visualize each table's relationship with one another. The database diagram of our example above would look like:

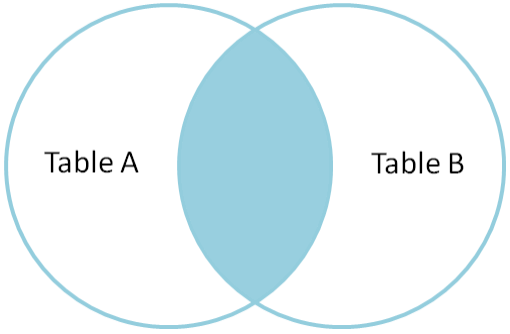


The table adjacent the gold key in the lines connected two tables implies that table contains the primary key and the table with the infinity symbol contains a foreign key that references the table on the other end of the connecting line. For example, Book contains a column, PublisherID, that is a foreign key that references the primary key of Publisher, PublisherID.

We have successfully eliminated all possible redundancy from our original version of the Book table. Understanding the concepts of normalization and how data is typically organized is essential to becoming proficient in SQL querying. This small, trivial example has helped us grasp the concepts of basic database design, primary and foreign keys, and – without you even realizing it – the underlying principles necessary for joins.

Basics of the INNER JOIN

A join, at the simplest explanation possible, is the process by which you can connect two tables together. There are four types of joins: INNER JOIN, LEFT OUTER JOIN, RIGHT OUTER JOIN, and FULL OUTER JOIN. An INNER JOIN can be thought of as the intersection of tables based on some key. In well-designed databases, an INNER JOIN (all join types, in fact), is defined by the connection of a foreign key from one table to the primary key in another table. Venn diagrams are often used to visualize what happens when joins are completed. With that in mind, below is the visual representation of the INNER JOIN:



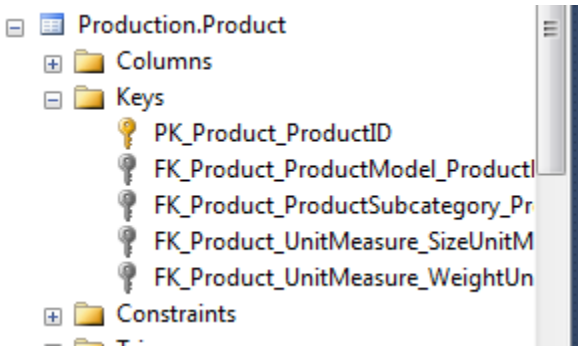
Notice how only the space shared by both tables is shaded in the diagram above. Let's work through an example of an INNER JOIN through a few queries and further explain the concept as we go.

Execute the query below in SQL Server Management Studio while connected to the AdventureWorks2012 database:

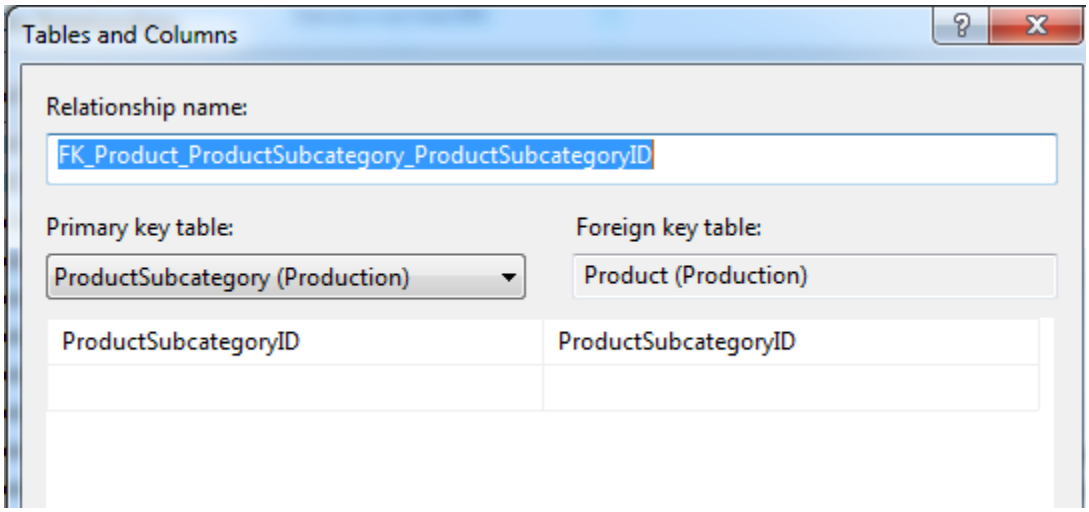
```
SELECT P.Name, P.ProductNumber, PS.Name AS ProductSubCategoryName
FROM Production.Product P
INNER JOIN Production.ProductSubcategory PS
ON P.ProductSubcategoryID = PS.ProductSubcategoryID
```

There are a few new concepts introduced here. First, you will notice the letters "P" and "PS" after the table names. These are called table aliases. They act similarly to column aliases in that you can use the alias to reference the table during your SELECT statement. Unlike the column alias, you can use this table alias in all clauses of the SELECT statement. Using table aliases when joining drastically improves readability of the code and minimizes unnecessary typing. The second thing you will notice is the form of the JOIN. After the FROM clause, and before any other clause, we specify the phrase "INNER JOIN" and then identify which table we would like to join to. In this case, we want to join the Product.ProductSubcategory table. After giving the Product.ProductSubcategory table the alias "PS", we type "ON" and then indicate how we are joining the two tables together. Knowing what column to join on is critical for success in joining tables together. In this example, the ProductSubcategoryID in the Production.Product table is a foreign key that references the ProductSubcategoryID column in the Product.ProductSubcategory table.

A valid question at this point is "how do you know which columns to join?" The object explorer is helpful in this instance. In the object explorer, expand the "Tables" folder under "AdventureWorks2012" if it is not already. Find the table named "Production.Product", and click the expand button to left of it. Find the toggle button to the left of "Keys" and click it.



The first key, the gold key, contains information about the primary key of this table. Well named keys often tell you exactly what columns are contained in the key. Double click on the second key named "FK_Product_ProductSubcategory_ProductSubcategoryID". Click the blank space to the right of the "Table and Column Specifications" option, and click the ellipsis button to the right of that. The "Tables and Columns" window will appear. This window details how this particular key is defined.



This table tells you that the ProductSubcategoryID in the Production.Product table (see the column on the right) is a foreign key with a relationship to the ProductSubcategoryID column of the Production.ProductSubcategory table. Given the foreign key relationship, joining on these columns is designed for this use.

Let's take a look at the query in our example. Run the query below and note the number of rows returned:

```
SELECT *
FROM Production.Product
```

The table Production.Product returns 504 rows. Now re-run our original query below and notice the rows returned:

```
SELECT P.Name, P.ProductNumber, PS.Name AS ProductSubCategoryName
FROM Production.Product P
INNER JOIN Production.ProductSubcategory PS
ON P.ProductSubcategoryID = PS.ProductSubcategoryID
```

This query only returns 295 rows. Why is this the case? Since an INNER JOIN is the intersection of two sets based on a column relationship, a row is returned only if the value in the joining column from one table matches at least one value of the joining column in another table. That is, a row will only be returned if the ProductSubcategoryID column's value in Production.Product appears in the ProductSubcategoryID column of the table Production.ProductSubcategory. Re-run the query:

```
SELECT *
FROM Production.Product
```

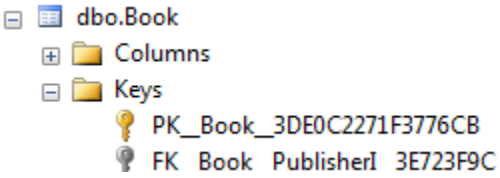
Once the results are returned, find the ProductSubcategoryID column and take a look at the values. You will notice a NULL value in many of the rows for this column. Since the value in those columns is NULL, that means that there is no value with which to join the row to in the Production.ProductSubcategory table. Since the INNER JOIN is an intersection of two tables based on the columns defined and there are many NULL values in our joining column from one table, those rows will not be matches to any row in the other table (in this case, the Production.ProductSubcategory table).

Let's go back to our example in the previous sub-section that discussed basic database diagram. After normalizing the initial Book table, we ended up with four tables: Author, Publisher, BookAuthor, and Book. If you have not already done so, download the SQL file that will create those tables in your AdventureWorks2012 database and execute the code http://knowlton-group.com/sql-training-class-resource/.

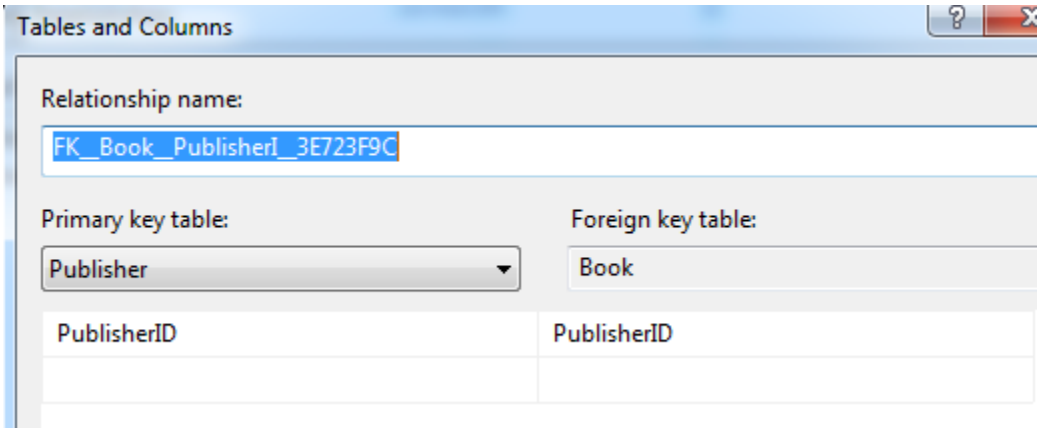
Run the query below to view the contents of the Book table:

```
SELECT *
FROM Book
```

Notice how the PublisherID is the last column. Expand the Book table item in the object explorer, then explain the Keys item as well (you may need to refresh the database if the items are not visible after running the code).



Double click on the second key. Once the "Foreign Key Relationships" window appears, click the ellipses to the right of the "Tables and Columns Specification" option. The "Tables and Columns" window should appear and contain the following contents:



As you can see, the PublisherID column in the Book table is referencing the primary key, PublisherID, from the Publisher table. If we wanted to join the Book and Publisher tables, we would use the PublisherID to make that connection.

Let's do that, in fact. We are going to create a SQL query that joins the Book table with the Publisher table through the PublisherID column in both tables. We will then be able to return the details of the book along with the name of the publisher. To do this, execute the query below:

```
SELECT B.Title, B.ISBN, B.PublishDate, P.PublisherName
FROM Book B
INNER JOIN Publisher P
ON B.PublisherID = P.PublisherID
```

Notice the use of table aliases "B" and "P" to represent the Book and Publisher tables, respectively. Our SELECT clause returns only the columns we wish to return. These columns include the details about the book, contained in the Book table, and then the name of the publisher that is contained in the Publisher table. We originate our query in the Book table as specified in the FROM clause. Next, we indicate that we are going to be utilizing an INNER JOIN. Then, we specified the table we wish to join to; in this case, the Publisher table is the table we are joining to. Next, we type "ON" which tells SQL that we are about to tell it how we wish to connect our tables. Lastly, we specify that we want to connect these tables when the PublisherID column from the Book table matches a PublisherID column in the Publisher table.

When viewing the results of the query, none of the IDs appear, only the columns that we have indicated. For your own benefit, execute these two queries below:

```
SELECT B.Title, B.ISBN, B.PublishDate, B.PublisherID, P.PublisherName
FROM Book B
INNER JOIN Publisher P
ON P.PublisherID = B.PublisherID

SELECT *
FROM Publisher
```

Notice how we include the PublisherID column from the Book table in the first query. Manually validate that the first query returned the proper value for the PublisherName column. Verify that each time a "1" appears in the PublisherID column of Book that it matches the PublisherName column value associated with the row in the Publisher table with a "1" in the PublisherID column. Do this for each book. This is a very simple way to validate that the join you executed is performing as intended.

Let's look at another example. In the AdventureWorks2012 database, there is table, named Person.Person, that contains some basic information about individuals in a database. There is another table, Person.EmailAddress, that contains the email address for each person in the Person.Person table. Both tables use the column BusinessEntityID as the key identifier for the individual. If we wanted to return the FirstName and LastName columns from Person.Person and the EmailAddress column from Person.EmailAddress, we would execute the query:

```
SELECT P.FirstName, P.LastName, E.EmailAddress
FROM Person.Person P
INNER JOIN Person.EmailAddress E
ON E.BusinessEntityID = P.BusinessEntityID
```

This query is telling SQL to look at each BusinessEntityID in Person.Person and find the row with the same BusinessEntityID in the table Person.EmailAddress. When that matching row is found, return the EmailAddress column. Because this join is an INNER JOIN, rows are only returned if the joining key is present in both tables; in this case, the joining key is the BusinessEntityID column.

Often times the data we need is spread across more than two tables. How can we use the INNER JOIN (or any join for that matter) to handle retrieving data from more than two tables in the same query? The process is nearly identical to joining between two columns. Suppose we wanted to take our previous example but add the person's phone number. The phone number is stored in the table Person.PersonPhone. Person.PersonPhone contains the BusinessEntityID column that we can use to join each table together. To add the phone number, execute the query:

```
SELECT P.FirstName, P.LastName, E.EmailAddress, PP.PhoneNumber
FROM Person.Person P
INNER JOIN Person.EmailAddress E
ON E.BusinessEntityID = P.BusinessEntityID
```

```
INNER JOIN Person.PersonPhone PP
ON PP.BusinessEntityID = P.BusinessEntityID
```

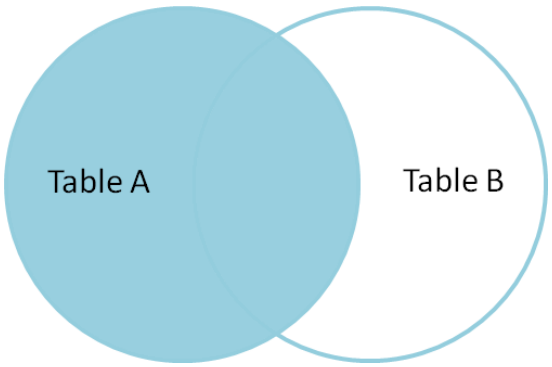
Since each BusinessEntityID has a single row in the Person.PersonPhone table, our row count remains unchanged in the results. Notice that all we needed to do was add another INNER JOIN line, specify the table we wanted to join to, and then specify which columns we would be joining on. Since there is a "one-to-one" relationship between each of these tables – that is, each row in a table matches up to only one row in another one of these tables – we can modify our join slightly. Instead of joining between Person.PersonPhone and Person.Person on the BusinessEntityID column, we can join between Person.PersonPhone and Person.EmailAddress:

```
SELECT P.FirstName, P.LastName, E.EmailAddress, PP.PhoneNumber
FROM Person.Person P
INNER JOIN Person.EmailAddress E
ON E.BusinessEntityID = P.BusinessEntityID
INNER JOIN Person.PersonPhone PP
ON PP.BusinessEntityID = E.BusinessEntityID
```

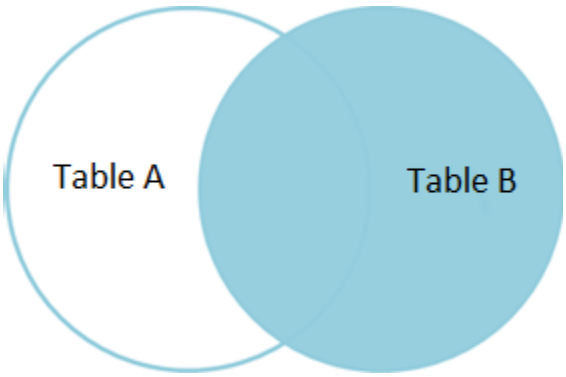
The concept of the INNER JOIN will become clearer with more practice. Given that, complete the following lab questions to further cement your understanding of this very important aspect of SQL querying.

LEFT OUTER JOIN and RIGHT OUTER JOIN

What is the LEFT OUTER JOIN and RIGHT OUTER JOIN? If you remember back to the introduction to INNER JOINs, it was explained that the INNER JOIN was the intersection of two sets. The LEFT OUTER JOIN and RIGHT OUTER JOIN can be thought of as "everything from one set and then information from the intersection of both sets". Let's visualize this with a Venn diagram for the LEFT OUTER JOIN:



And the Venn diagram for the RIGHT OUTER JOIN:



One thing to note is that the LEFT OUTER JOIN and RIGHT OUTER JOIN are nearly identical – the only difference is which set is completed shaded.

Looking at some examples will further clarify the LEFT OUTER JOIN and RIGHT OUTER JOIN. If you remember to the last section, we executed the query:

```
SELECT P.Name, P.ProductNumber, PS.Name AS ProductSubCategoryName
FROM Production.Product P
INNER JOIN Production.ProductSubcategory PS
ON P.ProductSubcategoryID = PS.ProductSubcategoryID
```

This query returns only 295 rows, yet the Production.Product table contains 504 rows. We discussed that since this was an INNER JOIN and since some of the column values in ProductSubcategoryID were NULL in Production.Product, then no rows could be matched to them from Production.ProductSubcategory. But suppose we wanted to return **EVERY** row in Production.Product and include the product's subcategory name (contained in the Name column from Production.ProductSubcategory) if one existed. The LEFT OUTER JOIN would solve this for us:

```
SELECT P.Name, P.ProductNumber, PS.Name AS ProductSubCategoryName
FROM Production.Product P
LEFT OUTER JOIN Production.ProductSubcategory PS
ON P.ProductSubcategoryID = PS.ProductSubcategoryID
```

Notice how there are many rows in the result set that have a NULL value in the ProductSubCategoryName column. This is fine and expected; these rows have a NULL value in the ProductSubcategoryID column in Production.Product, so it makes sense that they wouldn't match to row in Production.ProductSubcategory. If you imagine the LEFT OUTER JOIN Venn diagram while viewing the results, you can see that all rows are returned from Production.Product (Table A in the diagram) and only the values that match as part of the intersection defined by the join from Production.ProductSubcategory.

Now, execute this query:

```
SELECT P.Name, P.ProductNumber, PS.Name AS ProductSubCategoryName
FROM Production.ProductSubcategory PS
RIGHT OUTER JOIN Production.Product P
ON P.ProductSubcategoryID = PS.ProductSubcategoryID
```

Notice how we made the Production.ProductSubcategory table in the FROM clause and shifted the Production.Product table to the RIGHT OUTER JOIN line. Despite the change in code, the results are identical. Why? This is because the LEFT OUTER JOIN and RIGHT OUTER JOIN only differ in the order in which they are evaluated. The RIGHT OUTER JOIN returns everything from Table B and those values for rows that match in Table A. The LEFT OUTER JOIN returns everything from Table A and those values for rows that match in Table B. That is the **ONLY** difference between the LEFT OUTER JOIN and RIGHT OUTER JOIN. The LEFT OUTER JOIN evaluates the table listed first (on the "left") and then the table listed second (on the "right").

Take this generic form query:

```
SELECT A.[Column 1], A.[Column 2], B.[Column 3]
FROM [Table A] A
LEFT OUTER JOIN [Table B] B
ON A.[Primary Key] = B.[Foreign Key]
```

This query's results will contain all rows from Table A and then the [Colum 3] value from Table B where the primary key value from Table A matches to a foreign key's value in Table B.

Let's look at another example: suppose we wanted to identify the first and last name for each sales person associated with every single sale recorded in the Sales.SalesOrderHeader table. However, we want to include all sales regardless of whether or not a sales person was listed on the sale (in this database, sales that were placed online do not have a sales person associated with them). Return the SalesOrderNumber and TotalDue columns from Sales.SalesOrderHeader and the FirstName and LastName columns from Person.Person. To complete this query, you would execute:

```
SELECT SOH.SalesOrderNumber, SOH.TotalDue, P.FirstName, P.LastName
FROM Sales.SalesOrderHeader SOH
LEFT OUTER JOIN Sales.SalesPerson SP
ON SP.BusinessEntityID = SOH.SalesPersonID
LEFT OUTER JOIN Person.Person P
ON P.BusinessEntityID = SP.BusinessEntityID
```

Notice how we first needed to make a LEFT OUTER JOIN between Sales.SalesOrderHeader and Sales.SalesPerson. Looking at the keys of Sales.SalesOrderHeader, you will find a foreign key relationship between the SalesPersonID column of Sales.SalesOrderHeader and the BusinessEntityID for Sales.SalesPerson. Since we wanted to return all rows from the Sales.SalesOrderHeader regardless of whether or not a sales person was listed (and you will notice many rows in this table have a NULL value in the SalesPersonID column), we needed to use a LEFT OUTER JOIN to connect the two sets. Then, we want to use a LEFT OUTER JOIN to join the Sales.SalesPerson and Person.Person tables. We use the LEFT OUTER JOIN because not every row in Sales.SalesOrderHeader has a SalesPersonID, therefore not every row that is returned will have a BusinessEntityID matching. It then follows that since there will be a NULL value for the BusinessEntityID column value returned by the join of Sales.SalesOrderHeader and Sales.SalesPerson, we must use a LEFT OUTER JOIN to connect

Sales.SalesPerson and Person.Person. If we used an INNER JOIN, those BusinessEntityID values from Sales.SalesPerson would have no match to Person.Person and thus greatly reduce our result set.

There are scenarios where a query will require one LEFT or RIGHT OUTER JOIN and an INNER JOIN to complete a request. For example, suppose we wanted to return all rows from Sales.SalesOrderHeader that had a sales person listed on the sale. In other words, we only want to return those sales where the SalesPersonID column in Sales.SalesOrderHeader matches a BusinessEntityID column value in Sales.SalesPerson. However, we also want to view the sales territory that each sales person is associated with regardless of whether or not they have a listed sales territory. Since there are rows in the Sales.SalesPerson table with a NULL value in the TerritoryID column, we need to use a LEFT OUER JOIN from Sales.SalesPerson to Sales.SalesTerritory in order to complete this request. Putting it all together, we can complete the aforementioned request with the following query:

```
SELECT
       P.FirstName, P.LastName, T.Name AS TerritoryName,
       SOH.SalesOrderNumber, SOH.TotalDue
FROM Sales.SalesOrderHeader SOH
INNER JOIN Sales.SalesPerson SP
ON SP.BusinessEntityID = SOH.SalesPersonID
INNER JOIN Person.Person P
ON P.BusinessEntityID = SP.BusinessEntityID
LEFT OUTER JOIN Sales.SalesTerritory T
ON T.TerritoryID = SP.TerritoryID
```

Notice the use of the INNER JOINs between Sales.SalesOrderHeader and Sales.SalesPerson. This ensures that we only returns rows for sales that had a listed sales person. The join between Sales.SalesPerson and Sales.SalesTerritory is a LEFT OUTER JOIN. Had we used an INNER JOIN here, any sale for a sales person that wasn't associated with a sales territory would be excluded from our results. This would have caused our result set to be less than it should be.

What about using the other clauses while using a join? This is fairly simple; imagine that the joins are all part of the FROM clause. So, in the previous example, if we wanted to only returns those rows where the territory's name was "Northeast", we would execute:

```
SELECT
       P.FirstName, P.LastName, T.Name AS TerritoryName,
       SOH.SalesOrderNumber, SOH.TotalDue
FROM Sales.SalesOrderHeader SOH
INNER JOIN Sales.SalesPerson SP
ON SP.BusinessEntityID = SOH.SalesPersonID
INNER JOIN Person.Person P
ON P.BusinessEntityID = SP.BusinessEntityID
LEFT OUTER JOIN Sales.SalesTerritory T
ON T.TerritoryID = SP.TerritoryID
WHERE T.Name = 'Northeast'
```

And if we wanted to sort by the TotalDue column in descending order:

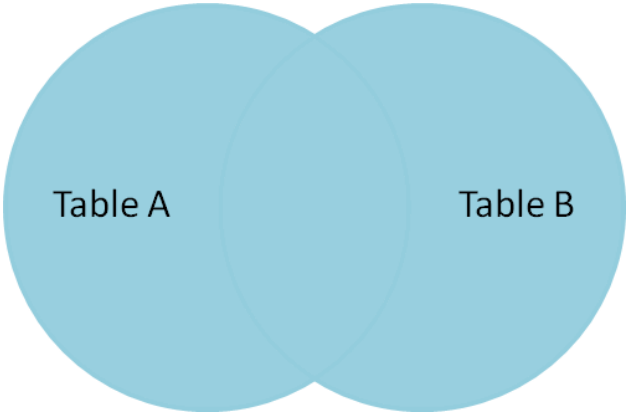
```
SELECT
       P.FirstName, P.LastName, T.Name AS TerritoryName,
       SOH.SalesOrderNumber, SOH.TotalDue
FROM Sales.SalesOrderHeader SOH
INNER JOIN Sales.SalesPerson SP
ON SP.BusinessEntityID = SOH.SalesPersonID
INNER JOIN Person.Person P
ON P.BusinessEntityID = SP.BusinessEntityID
LEFT OUTER JOIN Sales.SalesTerritory T
ON T.TerritoryID = SP.TerritoryID
WHERE T.Name = 'Northeast'
ORDER BY SOH.TotalDue DESC
```

The different clauses can easily be employed while using joins. Follow the same order that the general form requires. The only subtle difference is that the joins "belong" to the FROM clause. Becoming familiar with the general form of a SELECT statement when using joins and all clauses is critical to your success in querying.

The LEFT OUTER JOIN and RIGHT OUTER JOIN will become more easily understood with continued practice. Understanding when to use a LEFT OUTER JOIN or RIGHT OUTER JOIN as opposed to an INNER JOIN requires complete awareness of the request and the data within the database. Progress may be slow when querying a new database as you may not understand every single table's relationships. Be sure to consult the object explorer and identify the foreign keys. Feel free to write a SELECT statement that returns all rows from a table so that you know how many rows to expect in your results. Like most things: practice makes perfect.

FULL OUTER JOINs

We won't spend too much time on FULL OUTER JOINs as they are less frequently used than the other three join types discussed in this section. A FULL OUTER JOIN returns rows from both tables involved in the join regardless of whether or not a match is identified on the joining key. Visually, the Venn diagram of a FULL OUTER JOIN looks like:



Regardless of whether a match is found for the joining key, rows will be returned. So as to not overcomplicate the difficult that joins may initially present, we will not do anything

further with a FULL OUTER JOIN. If an example later on presents itself that requires the FULL OUTER JOIN, additional time will be dedicated to explaining its usage further. For now, simply be aware of its existence and basic definition.